CSC 211 Assignments 5 and 6

A big-integer class

BigInt

A class for storing very large numbers.

About

This class aims to reproduce an extremely small subset of features from the [GNU Multiple Precision Arithmetic Library][1], more specifically, the big integer class, and it's relative methods.

Big integers refer to numbers that are too large to be held in a computers memory. Specifically, for this assignment we are looking at unsigned integers which, as of this writing, can range from 0 to 4,294,967,295 on 32 bit systems; and 0 to 18,446,744,073,709,551,615 on 64 bit systems. Though this number may be large, there are occasions in computation where even larger numbers are needed.

For this assignment you will create a library capable of holding arbitrarily large numbers capable of basic arithmetic operations. It is a two-phase project, starting with construction and printing, ending with the arithmetic operations.

How do we go about holding arbitrarily large numbers? You have already implemented a **dynamic array**; this earns you the right to use a valuable part of the C++ standard library: the `std::vector` class. For this assignment, we will represent a bigint as the following type:

```
std::vector<vec_bin> number;
```

Which you can read as "a number is a vector of vec_bins" while a `vec_bin` is an integer type that will represent a **single digit** of a number. So we're going to represent numbers just like you learned in school: as a sequence of digits.

Below you can find a comprehensive overview of what is due during each phase of the project.

[1]: [https://gmplib.org/]

Getting Started

In your CS50IDE environment run the following command:

```
git clone https://github.com/csc211/bigint
```

This will give you a `compile` script, and some starter code for the `bigint` class. Note that there is a `main.cpp` that may look a little hard to understand. It uses the `catch` testing framework, and has a few test cases filled out. You are welcome to use this to build up your test cases to be sure your bigint class is implemented correctly.

Submitting

For both Milestone 1 and Milestone 2, you will submit your code to Mimir. All you need to submit is `bigint.cpp`, since we've already given you `bigint.h` and a `compile` script.

Milestone 1 - Due Thursday, November 8th by 12PM (before class)

For this milestone, you will get the construction, number access, comparison and display methods working.

After completing this milestone you should be able to build a `BigInt` from any source, as well as print out contents, examine the underlying structure, and test for equality between BigInts.

Constructors

Default constructor

`bigint a;` yields a `BigInt` equal to 0.

String Constructor

`bigint a("100000000");` yields a `BigInt` equal to the integer value of the string provided, in this case: 100,000,000.

Integer Constructor

`bigint a(100);` yields a `BigInt` equal to the integer provided, in this case: 100. Integers can be up to `unsigned long long` in size.

File Constructor

`bigint a(infile);` yields a `BigInt` equal to the integer value of the string stored in the `infile`.

Reading to and from files allows users to save their work to local memory, and resuming running arbitrarily large computations.

BigInt Constructor

```
bigint a(0);
bigint b(a); // Creates b from a.
```

Yields two equivalent `BigInts`, both equal to 0 in this case.

Yields two equivalent `BigInts`, both equal to 0 in this case.

Methods

to_string(bool commas = false)

Returns the string interpretation of the `BigInt`, with an optional flag to generate a string that utilizes commas for formatting.

```
bigint a = 1000000;
std::cout << a.to_string() << std::endl; // prints "1000000\n"
std::cout << a.to_string(true) << std::endl; // prints "1,000,000\n"
```

to_file(std::ofstream &outfile, unsigned int wrap = 80)

Writes a `BigInt` to a provided `ofstream`. The optional `wrap` parameter adds line breaks so that each line is only `wrap` characters long.

scientific(unsigned int decimal_points = 3)

```
bigint a(100000);
std::cout << a.scientific() << std::endl; // Should yield a scientific notation representat
```

This is used for examining extremely large quantities, in a human-readable format.

get_number()

Returns the vector representation of the number. No conversion is made from internal representation, this method is typically used for testing, and can be used for BigInt to BigInt construction.

(private) strip_zeros()

Used to remove leading zeros from a `BigInt`.

Operators

Digit Access: []

```
bigint a(123);
int x = a[2]; // x == 1.
```

Yields the digit at the specified index. This method is typically used for testing, and during internal arithmetic operations. Note that digits are stored in the underlying vector such that x[0] is the *least significant digit* (i.e. the 1's place) and `x[n]` (where n is the last element of the vector) is the *most significant digit*.

Yes, this might look backwards to you.

Comparators

Equality: == and !=

```
bigint a(10);
bigint b(10);
a == b; // Yields true.
a != b; // Yields false.
```

The equality operator works on `BigInt`s by examining the internal structures for digit by digit equality.

Milestone 2 - Due Thursday, November 15th by 12PM (before class)

Methods

add(bigint &that)

Returns a new `BigInt` which is the sum of `*this` and `that`. `*this` and `that` are not modified by this operation.

```
bigint x = 10;
bigint y = 15;
bigint z = x.add(y); // z is 25, moreover x is still 10, y is still 15.
```

subtract(bigint &that)

Returns a new `BigInt` which is the difference of `*this` and `that`. `*this` must be larger than `that` for this operation to succeed (negative values are not allowed).

```
bigint x = 500;
bigint y = 800;
bigint z = x.subtract(y); // NOT ALLOWED - Throws error.
bigint z = y.subtract(x); // Valid, z = 300; y = 800; x = 500;
```

multiply(bigint &that)

Returns a new BigInt which is the product of `*this` and `that`. `*this` and `that` should not be modified by the operation.

```
bigint x = 5;
bigint y = 2;
bigint z = x.multiply(y); // z = 10; x = 5; y = 2;
```

divide(bigint &that)

Returns a new BigInt which is the quotient of `*this` and `that`. `*this` and `that` should not be modified by the operation.

```
bigint x = 6;
bigint y = 2;
bigint z = x.divide(y); // z = 3; x = 6; y = 2;
```

mod(bigint &that)

Returns a new BigInt which is the remainder of the division between `*this` and `that`, both of which should not be modified by this operation.

```
bigint x = 11;
bigint y = 10;
bigint z = x.mod(y); // z = 1; x = 11; y = 10;
```

pow(unsigned long long n)

Raises a `BigInt` to the given power, `n`, directly modifying the `BigInt`.

```
bigint x = 2;
x.pow(2); // x = 4;
```

Operators

Addition: +, +=

Adds two `BigInts` together.

```
bigint a = 1000;
bigint b = 1e50;
bigint c = a + b;
c += a;
c += b;
```

Subtraction: -, -=

Reduces one `BigInt` by the other, throwing an error if the latter is larger than the former. The base implementation of this class does not permit negative values.

```
bigint a = 1000;
bigint b = 1e50;
bigint c = a - b; // Would throw an error
bigint c = b - a; // valid
c -= a; // valid
c -= b; // throws an error
```

Multiplication: , =

Multiplies two `BigInts` together.

```
bigint a = 10;
bigint b = 1e27;
bigint c = a*b;
c *= a;
```

Division: /, /=

Performs floor division on two `BigInts`.

```
bigint a = 5;
bigint b = 2;
bigint c = a / b; // c = 2
c /= 2; // c = 0
```

Greater Than / Less Than: >, >=, <, <=

Compares the values of two `BigInts`, returning true if the former is larger.

```
bigint a = 10;
bigint b = 20;
a > b; // Returns false.
a >= b; // Returns false.
a < b; // Returns true.
a <= b; // Returns true.
```

Modulus: %, %=

Returns the remainder of the division between two `BigInt`s.

```
bigint a = 11;
bigint b = 3;
bigint c = (a % b); // c = 2
```

Stream operator, <<

`BigInt` provides stream operators for ease of printing, the `ostream` (`<<`) operator runs `to_string`.

Grading Rubric

For this assignment, correctly passing all tests on Mimir is worth 80% of your grade. The remaining 20% is based on reasonable commenting habits, and the structure and organization of your code.

Submitting

You will submit `bigint.cpp` via Mimir, where its functional correctness will be graded automatically. For this assignment, you only get 5 submissions to Mimir, so make sure to **test locally** first.